webOS: Palm's Game-Changing Mobile Operating System

Learn about Palm's exciting new webOS and how to get started developing for it

Frank W. Zammetti fzammetti@etherient.com http://www.zammetti.com

The Apple iPhone ushered in a new age where the "smartphone" rules the mobile roost. Apple has lots of competition in the space, of course, and Palm is a big one having recently unleashed upon the world what could wind up being an absolute game-changer: webOS. In this article you'll learn what webOS is and how to develop for it (hint: you're reading a magazine on JavaScript and web development!)

Smartphones: The Future Now

For the first time in history, reality has out-paced Star Trek!

Do you remember the communicator? It was quite a marvel: communicate with anyone virtually anywhere on the planet via an orbiting spacecraft. Pretty amazing in the 60's and 70's, but all too commonplace nowadays!

Do you remember the other important piece of hardware they carried with them? No, not the phaser, although that'd be awesome if it were real! No, I mean the tricorder.

So, let me get this straight ... Star Trek is the future, yet in it they still need to carry two devices to get things done?

How old-fashioned!

You see, in the boring old here and now, we've managed to one-up the men and woman of the U.S.S. Enterprise. These days, we have a single device that can do it all, namely the smartphone. We can communicate with our friends and family, in multiple ways actually; we can pull up data at a moment's notice, we can take pictures, and in some cases can even scan for various types of radiation (audio, infrared in some cases).

We've actually managed to one-up the future here! How cool is that?

The Internet: Is That Thing Still Around?

A smartphone, as cool as it is on its own, wouldn't be quite what it is today without that other great invention of the 20th century: the Internet.

The Internet has many things going for it of course (and here I am not referring to sexyminxesofbeverlyhills.com). One big advantage is the relative ease with which people can create web sites and applications. HTML, CSS and JavaScript are, in many ways, a lot easier for people to wrap their brain around than things like C, C++ and, for those of us that liked to torture ourselves years ago, Assembly.

It wasn't long before someone pulled the old peanut butter and chocolate gag on smartphones and web technology and created the most delicious Reese's Peanut Butter Cup ever: webOS and the Palm Pre.

Palm was an early pioneer in the area of PDAs, and to some extent, smartphones. But, it can hardly be argued that they got lapped by their competitors, chiefly Apple. In fact, just a short time ago, many thought Palm as a company was in trouble of disappearing. They needed something to change their fortunes, and as luck would have it for them, the rise in popularity of the smartphone, in conjunction with a brilliant idea on Palm's part, provided just what they needed.

The folks at Palm had a genius, and yet in many ways obvious, idea: as powerful and connected as modern smartphones are, why would developers want to program in a way that is largely contrary to that? Why would a device that "lives in the cloud" have to be programmed with technologies that are platform-specific and, in some cases, even device-specific? Wouldn't developers, instead, want to use the same technologies and techniques they are using everywhere else, namely web technologies? The answer for most of us these days is a resounding "Yes, please!". Palm, to their credit, didn't just have a good idea, they acted upon it!

That, in a nutshell, is what webOS is all about.

Say hello to my little friend: webOS

Palm created webOS, it is what sets the Palm Pre smartphone apart from its competitors and what many people feel is a revolutionary achievement. One of the main things that sets it apart is that webOS is not tied to the Pre. WebOS can run on other devices just

fine (the Palm Pixi as an example), which is the opposite of what Apple has done with the iPhone, where the phone and operating system are linked at the hip.

The biggest thing however that separates webOS from the rest is that developing an application for webOS is writing a web application and nothing more. You write HTML, you write CSS, and you write JavaScript. WebOS provides a rich set of APIs and services that allow you to interact with the device from your JavaScript code. All the Ajax goodness you've picked up over the last few years applies 100% to webOS development.

Why is this model revolutionary and so important? Simply stated, it's because it makes the barrier to entry far lower than competing platforms and allows developers to reuse the knowledge they've built up over the last 10 to 15 years or so.

Another revolutionary aspect to webOS is that it is built from the ground up for "life in the clouds", that is, data storage and even program execution out in the cloud... the Internet, in other words! Things like alerts from remote sources, data updates, and remote synchronization are all considered core parts of the user experience, and as such are dealt with in a fundamental way by the OS. These aren't just tacked-on afterthoughts as it seems sometimes with other platforms.

All of this means that webOS gives you the power of native applications while building upon the strengths of the web development model. At the same time, it provides a richer experience that is focused squarely on the concerns of modern mobile users.

If you're interested in more technical details, here's a juicy one: webOS is a Linux derivative based on the 2.6 kernel. It uses the standard driver architecture managed by udev, and it utilizes its own proprietary boot loader (there is, as I'm sure you can guess, a good mix of Palm-provided components as well as open source goodness mixed into webOS). Two file system partitions are present: a private ext3-based partition for internal use and a FAT32-based "media" partition for your own storage. The later can be mounted via USB as an external storage device to make transferring files back and forth painless. Media files are handled via gstreamer, and this includes support for numerous codecs, for both audio and video. Playback can be file or stream based.

You can see the overall structure of webOS in Figure 1. It's not a complex model by any stretch, but this should help you visualize where the code you'll be writing lives. You generally will only have to think about the Mojo Framework section and the On-Device Services section, the rest will be in the background and not something you typically have to worry about, but they're there, making everything work for you!

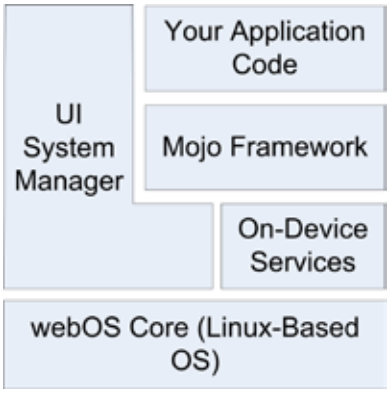

Figure 1. High-level structure of the components that make up webOS.

How Palm Got Its Mojo Back

When you write an application for webOS, you'll be talking to what's called the Mojo framework. With most other smartphones, you are coding directly to an API provided by the operating system and sometimes even going directly at the hardware itself, most commonly via C/C++ or Java. A webOS application, however, is different in that it is essentially running within a runtime on top of webOS, namely, a web browser. (More specifically, Palm calls this runtime the UI System Manager and it is built on top of WebKit, the engine that powers Safari, Chrome, and other web browsers too.) You won't, by just looking at it, know you're running within a web browser because the operating system effectively is the web browser and hides this from you (in the sense that you won't see typical web browser "chrome" elements like buttons and toolbars and such), but that's basically what is happening. The result of this browser based UI System Manager approach is that we get to code a webOS application using standard web technologies like HTML, CSS, and JavaScript.

The Mojo framework is how you interact with webOS. It is a JavaScript framework that is conceptually split into three parts: the APIs, services, and widgets:

The APIs are JavaScript classes and functions you make use of from your code. These come in JavaScript packages that organize their functionality logically.

Services are essentially local (meaning running on the Pre) servers that you can access using Ajax-like techniques that provide somewhat lower-level access to features of the Pre (things like playing audio and launching a true web browser, for example). Because these servers are local, you don't have the same sorts of latencies and response times you see with a typical Ajax application, but that essentially is what you're writing when you use these services (even to the extent that you call them asynchronously).

Widgets are of course the UI elements that you'll use to build your applications. While you aren't required to use them, strictly-speaking, they are to a large extent what makes webOS so powerful, so you'll want to use them or sure!

Mojo, being a proper framework, provides more than just APIs for you to call; it also provides a prototypical architecture for your application to follow. Much of a webOS application is based around the notion of conventions; that is, as long as you follow certain guidelines and structural recommendations, there will be less code to write because Mojo will know how to execute your application intrinsically. It removes a lot of plumbing-type code from having to be your responsibility, which is a very good thing for us generally lazy developers!

The structure of a webOS application is based on the well-known Model-View-Controller (MVC) pattern (shown in Figure 2), which provides separation between various concerns in your application, namely, the parts responsible for displaying information, the parts responsible for implementing the underlying logic of the application, and the parts dealing with the data the application works on (as well as the data itself). This structure maps to the default locations where you'll put various parts of the code that make up your application, and by following this pattern, Mojo knows where to find the parts of your application it needs by default and knows how to make use of those parts.

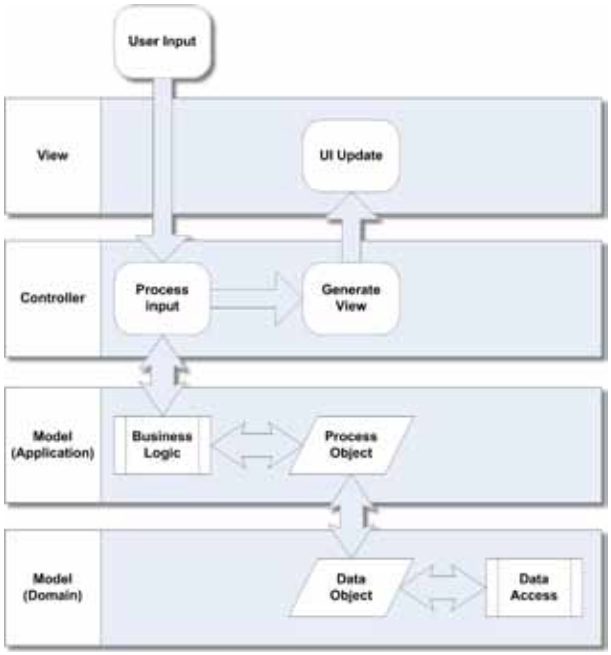

Figure 2. The MVC architecture, generalized for UI interactions

webOS Standard Application Structure

Figure 3 shows the basic directory structure that most webOS applications will generally take. While you can deviate from this structure if you like, following the code-by-convention model provides enough benefit that you generally should stick to this structure.

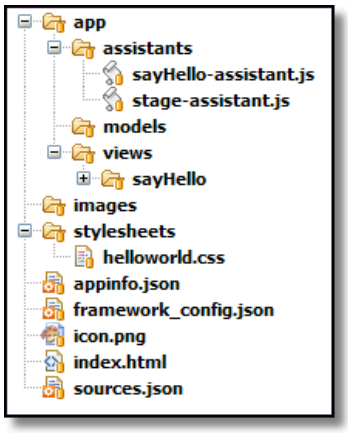

Figure 3. Basic directory structure of an app

At the root of the directory structure you will find a number of files:

- index.html (required), which is the HTML for your stage (we'll get to what that is shortly).
- sources.json (technically optional, but usually present), which lists the JavaScript files that make up your application.
- appinfo.json (required), which is some meta-information that describes your application to Mojo.
- A required icon that represents your application.

In the root directory, you'll also find an `app` directory, and this is where the code of your application lives.

As a web application, you may have images, scripts, and style sheets, and by convention you'll find them in directories named `images`, `javascripts`, and `stylesheets`, right off the root (again, you can put them anywhere you like, even if it's unusual).

In the `app` directory you'll find some subdirectories including `assistants`, `models`, and `views`. The `assistants` directory contains JavaScript files that control scenes (which we'll also get to shortly), and the `views` directory contains HTML files corresponding to a given scene. The `models` directory is frequently not needed but is where application data models can reside (in my experience it is more common to not have anything here).

As I alluded to earlier, the parts of an application fall into some basic categories that are built into the directory structure, these being views and assistants primarily. Before we can really talk about these things, though, we need to cover two basic concepts: stages and scenes.

Stages

A stage is a container for your application, much like the tab of a tabbed web browser can be considered a container for a web site. A stage is conceptually represented by a card in the UI, and an application can have more than one stage, or card, opened, if necessary, although one is frequently sufficient.

A stage is also an HTML document that serves as the foundation that the scenes of your application are shown on. The HTML document representing a stage tends to be pretty sparse, and most of the time it has no content that the user sees. The stage can usually be pretty bare because it's in the scenes where most of the action tends to take place. You'll see later that, for a typical application, there is a single HTML document that is initially loaded and effectively serves as the stage for the application. This document, in the absence of any scenes, also serves as the view of the stage. You could, in fact, construct an entire webOS application with nothing but this single stage HTML document serving as its view, but that's a bit atypical because scenes are what you'll usually use to present a view of an application.

Scenes (Views and Assistants)

A scene is a view into your application. Like a stage, it is built from an HTML document, but tied to that document is a scene assistant. This is simply a JavaScript object that aids the controller for the scene. Mojo will spawn a controller, a JavaScript object, for the scene automatically, but most of the functionality behind the scene is provided by the assistant you program (hence the name: it assists the controller).

Remember MVC? Well, the HTML document for the scene is your view, while the scene assistant is your controller (for all practical purposes it is, even though it's not truly the controller). The model might be some data stored in a local database or a remote system that provides data, or you may not even have a model at all in some cases.

The HTML document that makes up a scene's view is in fact a fragment of HTML, meaning it's not a complete document with a `<head>` and `<body>` and all that. For example, Listing 1 shows a perfectly valid scene HTML document.

```
<br>
<div id="myButton" x-mojo-element="Button"></div>
```

Listing 1: a valid scene HTML document

Now, back to scenes! An application usually has at least one scene, and most applications will tend to have quite a few more. You can conceptually think of a scene as different pages of a web site that the user navigates between.

The Scene Stack

When the user navigates from one scene to another, the previous scenes are not necessarily lost. Each time a new scene is shown, it is pushed onto a scene stack. The previous scenes sit below the current one on the stack. If a given scene is closed, it is popped off the stack, and the scene below it, if any, is again shown. This again is very much like navigating a web site when you use the back button of your browser.

Your application's code is indirectly responsible for the stack in the sense that your code will be pushing scenes onto the stack to show them, and sometimes it will pop the current one off to show the previous one, although a scene can also be popped by the user if they use the back gesture, a basic concept in the webOS UI.

Application Life Cycle

A webOS application, obviously enough, begins by being installed on your device! An application gets installed by going to Palm's App Catalog, as shown in Figure 4, which is the online store where you can browse, download, and purchase webOS applications. When you select an application, it will be automatically installed and added to your launcher for you. It's that easy!

Figure 4. The Palm App Catalog

Developers can also install software directly with tools that come as part of the SDK, but for an end user, it's the App Catalog that is the starting point (there are also "homebrew" channels where you can get software without going to the App Catalog, but that's a topic for part II of this article).

Once an application is installed, Mojo and webOS also provide your application with a well-known runtime life cycle, or series of events that occur at prescribed times when your application is executed. The first step is that Mojo will read the `appinfo.json` file. This provides the operating system with vital information about your application.

Next, the app controller will be executed and will delegate to your provided app assistant. This is a JavaScript class you provide that can handle things such as arguments passed to the application at launch and initial stage setup. You usually will not need an app assistant; they come into play most frequently with background, or headless, applications. That's an advanced topic and won't be covered here.

Part of the `appinfo.json` file is the name of the HTML file that serves as the "launch point" for the application, typically `index.html`. Next, that file is loaded and parsed. Any style sheets and JavaScript files referenced in the HTML file (or those listed in `sources.json`) will then be loaded and parsed. This HTML document is responsible for loading the Mojo framework itself and, in the absence of any views being pushed, serves as the view for the stage as well.

Next, Mojo invokes methods on the stage controller, which is a JavaScript object that provides control-layer functions for the stage itself. You provide a stage assistant that the controller delegates calls to in many instances. The first such method invoked, following the constructor of the assistant class itself, is the `setup()` method. What you do in this method is entirely up to you, but the most typical activity is to push the first scene onto the scene stack, causing it to be shown. Like the stage, the scene you push will have a controller, as well as an assistant, and that assistant also will have a `setup()` method. The `setup()` method will be called only when the scene is pushed, not when the user uses the back gesture to return to it.

The `activate()` method of the stage will next be called. You don't even have to implement a method of a controller in your assistant that you don't need, Mojo will be just fine without it because there is a default implementation of all of them in the controller, and often-times you won't need an `activate()` method at all. The `activate()` method can be called multiple times, any time the scene is brought into view in fact, either because it was pushed or because a scene on the stack above it was popped.

Speaking of popping a scene, any time a scene is popped off the scene stack, the deactivate event will fire, and your scene assistant's `deactivate()` method will be called.

Getting started with webOS Development

If all of this sounds great and you want to get into this webOS thing and code some apps, you're in luck because all you need is 100% free! Palm provides an SDK that anyone can download and which provides all the basic tools you'll need. In addition to the SDK you'll need to have a Java 6 runtime installed first. You can get that at http://java.sun.com. Grab the latest version, and install it on your workstation (Windows, Mac or other *nix variants). You'll also need to install VirtualBox, which is an x86 virtualization package from Sun that you can find at http://www.virtualbox.org. This will allow you to run the Palm Pre emulator that comes along with the SDK, so you don't even need to rush out and spend the money on a real phone, you can start without one just fine!

The webOS SDK can be downloaded from the Palm Developer Network site: http://developer.palm.com. This site is your main portal into webOS development where you can find useful articles, reference materials and community forums to help solve problems you'll encounter along the way.

Once you have Java, VirtualBox, and the SDK installed, you can launch the Pre emulator and start playing with webOS.

A Proper IDE Makes Things a Lot Easier

While the SDK alone is all you really need because you can compile, package and deploy webOS application with it from a command line, we're living in a more enlightened age, and an IDE can make things a bit more pleasant.

If you're already an experienced Eclipse user than I'll save you some time: head on over to http://developer.palm.com/index.php?option=com_content&view=article&id=1639 where you'll find a plugin install URL to use inside Eclipse.

If you've never heard of Eclipse, it's an IDE that provides, among other things, an extension mechanism via plugins to expand its capabilities. Palm kindly provides a plugin for doing webOS development. So, since the rest of this article assumes you're using Eclipse, head on over to http://www.eclipse.org, download the appropriate Eclipse package (which you chose shouldn't matter for doing webOS development) and install it. When you're done, launch it and use the Install New Software option on the Help menu. You'll need the URL mentioned earlier. Once you point Eclipse at that URL it should see the webOS plugin and ask you if you want to install it. Do so, and you'll be ready to rock and roll!

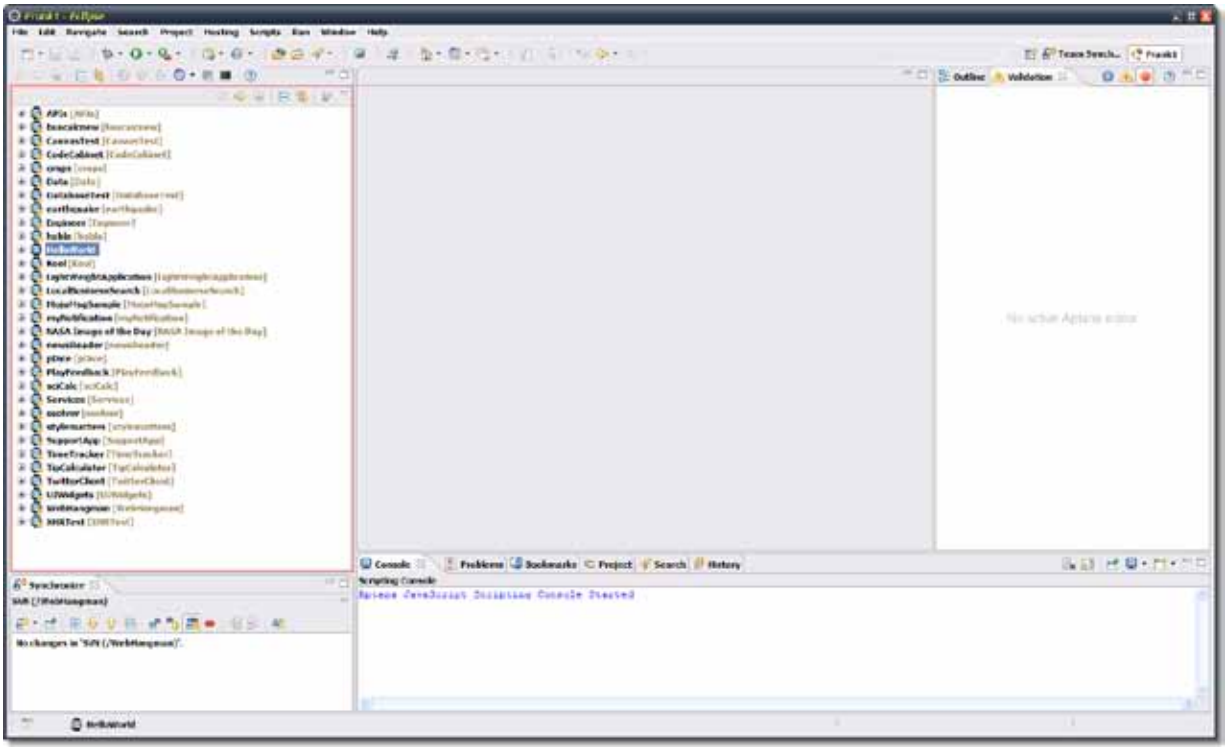

Figure 5. The Eclipse workbench

Building a Better (Hello) World

Assuming you've gotten all the installations out of the way, let's build ourselves a simple application. If you are already familiar with Eclipse, then much of this will be superfluous, but to be sure as many readers as possible are served by this, I'll assume that you have little or no experience with Eclipse.

The first step is to fire up Eclipse and if this is your first time running it you'll be asked for a workspace, which is just a fancy way of saying a directory where your code will live.

Once Eclipse starts, assuming this is a newly created workspace, you'll find yourself at a nice welcome screen. You can close this immediately, which will reveal your workbench, as shown in Figure 5. The workbench is presented as a perspective, which is just a configured collection of tools and settings that are specific to the technology you're working with (Eclipse supports multiple development technologies). There is a webOS perspective, but since you're likely to tweak whatever perspective to use to your liking, it's not terribly important that you use the webOS perspective.

Note that Figure 5 will look a bit different for you because I've already done some customization of my own environment, plus I have a bunch of projects in my workspace.

Anyway, let's get started! We'll create a new project by selecting the File menu and then the New option. You'll see a list of things you can create. You'll need to choose Other at the bottom, which will bring you to a pop-up dialog box, as shown in Figure 6.

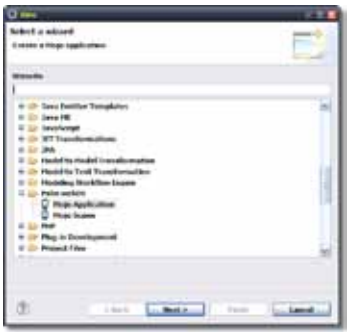

Figure 6. The first step of creating a webOS application in Eclipse

Select the Mojo Application project type, and click Next. At that point you'll be presented with a second dialog box where you can enter a few pieces of information that describes the project, as you can see in Figure 7. You only need to touch the Project Name field, the defaults for the rest are fine for our purposes here.

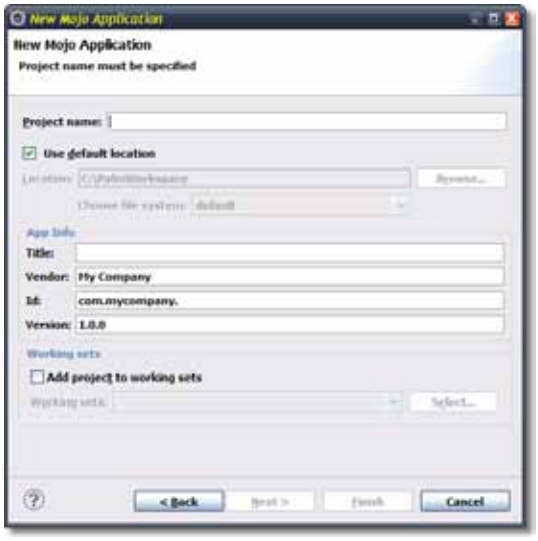

Figure 7. Entering a name for the new project

After a few seconds of work by Eclipse you'll see a new project on the left in the Project Explorer tab. If you expand this project, you'll find that its structure, and the files contained within it, matches the structure we looked at earlier. What you have here is a full, working, albeit simple webOS application!

In fact, let's see it in action now! Launch the Pre emulator by using the icon installed as part of the SDK. Once it's up, go back into Eclipse and right-click the project in the Project Explorer, select Run As, and select Palm Application. You'll be greeted with a pop-up asking you to specify the run target. Select the Palm Emulator option and click OK. A few seconds later, in the emulator, you should see what is shown in Figure 8.

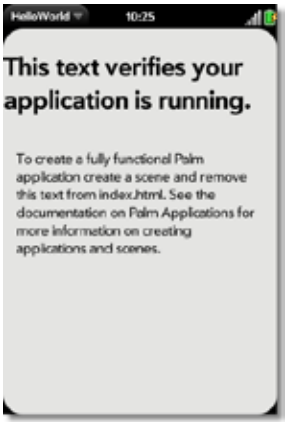

Figure 8. The basic application skeleton: it's alive!

Expanding the App

Let's make this app a bit more interesting now by adding a single scene to it, and on this scene let's add a text field the user can type their name into and a button. When the button is clicked, we'll display an alert message saying "Hello, XXX" where XXX is what the user types in the text field.

To create the scene, simply right-click on the project in the Project Explorer view, select New, and the Other. Go to that same Palm group where you went to create the project and this time select the "Mojo Scene" option. Simply enter `sayHello` as the name of the scene and the plug-in will go off and do the steps required to add a new scene. What Eclipse then does for us is first to create a new directory under the existing `views` directory named `sayHello`, and in it create a file named `sayHello-scene.html`. Second, a file named `sayHello-assistant.js` is added to the `assistants` directory. Finally, in the `sources.json` file, a reference to your scene's assistant is added.

The rest is up to us! First, we need to create the view for our new scene. The file we need to edit first is the scene's HTML file, so open `sayHello-scene.html`, and put the content in Listing 2 into it.

```
<br><br>
<div id="txtName" x-mojo-element="TextField"></div>
<br><br>
<div id="btnGreet" x-mojo-element="Button"></div>
```

Listing 2: The sayHello-scene.html file.

This is really just some plain old, boring HTML. The only interesting bit is that the `<div>`s have a special `x-mojo-element` attribute on them. This is the key bit because it tells Mojo, when it parses this document, to create some UI widgets for us. Essentially, the `txtName <div>` will be magically transformed into a `TextField` widget that allows the user to enter information, and the `btnGreet <div>` will be morphed into an actual `Button` widget.

The next step is to put some code into the scene assistant. Most importantly, we need to provide a model for those two UI widgets. Yes, even at the widget level the MVC architecture is in play! To do this, we'll need to edit the `sayHello-assistant.js` file and put the contents shown in Listing 3 into it:

```
function SayHelloAssistant() { };

SayHelloAssistant.prototype.txtNameModel = { value : "" };

SayHelloAssistant.prototype.setup = function() {

  this.controller.setupWidget("txtName",
    { maxLength : 15 }, this.txtNameModel
  );

  this.controller.setupWidget("btnGreet", { }, { label :
"Greet Me" } );
```

```
  Mojo.Event.listen(this.controller.get("btnGreet"), Mojo.
Event.tap,
    this.greet.bind(this)
  );

};

SayHelloAssistant.prototype.greet = function() {
  this.controller.showAlertDialog({
    onChoose : function(inValue) { },
    title : "Greetings!",
    message : "Hello, " + this.txtNameModel.value,
    choices : [
      { label : "Ok", value : "" }
    ]
  });
};
```

Listing 3: The SayHello scene assistant.

Now, this needs a bit of explanation. Simply stated, a scene assistant is a JavaScript class, which means a function. So, the first line creates that function, and once again, "configuration by convention" is in play here, so the name of the class has to be the name of the scene, with the first letter capitalized as well.

Next, we have to add some members to the class, which we do via the `prototype` of the newly created class. The first member is `txtNameModel`, which is the model that will be tied to the `TextField`. This is a simple object defined with JSON with a single field, `value`, which not surprisingly is the value of the text field.

Next, the `setup()` method is added to the assistant, and this is where, primarily, widgets are set up. Although the scene's HTML declares the widgets, and they will be created when the app is launched, there is some code that goes behind them as well, including their configuration. Before we get to that though, I need to point out what `this.controller` is: it's a reference to the scene controller for the current scene. Mojo spawns a controller for the scene automatically, which then delegates calls to your scene assistant. The controller itself provides functionality that the code in your assistant will need access to; hence, a reference to it is automatically added to your assistant so you can refer to the controller.

The first time we see this used is the first line of code in the `setup()` method. The call to `this.controller.setupWidget()` is one of the most frequent lines of code you'll write, over and over again! This is what configures a widget. This particular line is setting up the `TextField`. The first argument is the ID we assigned to the widget, `txtName` in this case.

The next argument is a bit of JSON that defines attributes of the widget. Each widget will have a different set of attributes, most optional. Here, I decided to limit the number of characters the user can enter to 15 by setting the `maxLength` attribute accordingly.

The third argument to the `setupWidget()` method is the model for the widget. Once again, what goes into the JSON object will depend entirely on the widget you are configuring. Here I've simply given it a reference to the `txtNameModel` field of the assistant class.

Following that is another call to `setupWidget()`, this time to set up the `Button`. In this case, there are no configuration attributes, so we have an empty object passed as the second argument. The third argument is the model for the button, and this time we have an attribute called `label`, which sets the text label shown on the button. Note that as opposed to the `TextField`, the model object is defined inline with the call to `setupWidget()`. Although I personally prefer this syntax, it does have some negative implications for memory utilization, so it's generally better to declare your model (and even config object) independently. This will be necessary if you need to modify the model after the fact, to change the label of the `Button` for example.

Next we see a call to `Mojo.Event.listen()`. This is another of the most common methods you'll be using time and again, and its purpose is to set up an event handler on a previously created widget. The first argument to this method is a reference to the widget you want to hook up an event listener to, which is gotten by a call to `this.controller.get()`, passing it the ID of the button. Note that `Mojo.Event.listen()` must be called after `setupWidget()` so that there's actually a widget created to attach the event to.

The second argument to `Mojo.Event.listen()` is the type of event to listen for. The event types are defined as "constants" and are defined in the `Mojo.Event` package. In this case we're listening for a tap event, so `Mojo.Event.tap` is what we want. The third argument is a reference to a callback function that will be called when the event occurs.

The function we're calling here is a method of the scene assistant class, namely, the `greet()` method. So, it might seem like just passing `this.greet` would have been sufficient, and in fact in some situations it would be, but it's not in this case. You see, the problem you encounter is that when the event occurs and the specified function is called, the meaning of the keyword `this` won't be what you expect, which is almost certainly that it refers to the scene assistant instance. Now, if your callback function doesn't actually need the `this` keyword, then you can get away with syntax like `this.greet`.

But, in the callback code here we are using the `this` keyword to get the value of the `TextField`, which is in the `value` attribute of the `txtNameModel` field. So, we need to ensure the `this` keyword has the appropriate scope and we do this by calling the `bind()` method on the function.

This is a method provided by the Prototype JavaScript library, which is included with the Mojo framework. The `bind()` method provides a specific context to the keyword `this` when the function

is executed. For example, say you have a function called `myFunction()`. What does the keyword `this` point to when the function is executed? In JavaScript, the answer depends on what the function is bound to at runtime. Since every function is in fact a property of some object ("naked" objects in the global scope are actually properties of the `window` object), the keyword `this` may not point to what you expect when the function is executed. Many times you need it to point to a specific object, which is where `bind()` comes in. If you do `myFunction.bind(X)`, where `X` is some object, then when you execute `myFunction()`, the keyword `this` will point to `X`, regardless of what it might have pointed to without using `bind()`.

The last bit of code in the assistant is the `greet()` method itself. This uses another method available on the scene controller, `showAlertDialog()`, which pops up a dialog from the bottom of the screen where the user must perform some action. This method is passed a JSON object that defines the dialog box. The first attribute in this object is `onChoose`, which is a function called when the user picks one of the choices available to them. Here there's nothing to do so an empty object is passed. Next is the `title` attribute, which provides the title that will be shown on the dialog box. The `message` attribute is next and is the text of the message shown on the dialog box. Here we concatenate the value entered into the `TextField` by grabbing `this.txtNameModel.value`. Mojo has taken care of updating the model object based on the user input for us. Finally, the `choices` array is a list of buttons that the user can click. Here there's only a single OK button. Each button has a `label` attribute, which is the text displayed on the button, and `value`, which is the value associated with the button and which will be passed to the `onChoose` function.

If you were to run the application at this point you'd find that it doesn't actually do anything different than it did initially. The reason is that there is one more thing we need to do, namely to show the `sayHello` scene. This is accomplished by "pushing" the scene, which means pushing it on top of the scene stack. This is fortunately easy to do: edit the `stage-assistant.js` file in `app/assistants` that was created by default and add to its currently empty `setup()` method the code in Listing 4.

```
this.controller.pushScene("sayHello");
```

Listing 4: Pushing the scene

Just like with our scene assistant, we have a `setup()` method for the stage, and also just like the scene assistant, Mojo adds a reference to the controller for the stage to our assistant automatically. This controller exposes a `pushScene()` method, which is what we need to get our scene showing. We pass the name of the scene here, and that's all it takes because we've followed the naming conventions and Mojo knows exactly how to show our scene. It gets pushed onto the scene stack, `setup()` is called, and the application does what we expect.

The result of all this effort once you run the application (and assuming your name is Frank too!) is what you see in Figure 9.

Figure 9. HelloWorld in all its final (ahem) glory

I'm the first to admit we're not going to win any awards with this application, but it definitely gives you the basic foundation you need to begin writing webOS applications.

Coming soon: webOS, Part Deux

In next month's issue, we'll continue exploring webOS. Specifically, we'll look at more of the functionality provided by Mojo. We'll look at the UI widgets available as well as the on-device services that give you more "direct" access to the hardware and the somewhat lower-level capabilities of the device. See you back here, same JavaScript time… same JavaScript channel!

If you're in a rush and want a lot more information, you could always (shameless plug alert!) purchase my book, Practical Palm Pre webOS Projects from Apress. You can get info at http://www.zammetti.com/booksarticles

---

Frank W. Zammetti has been developing software for over a quarter century in a wide variety of technologies for myriad platforms. Frank has been a professional developer/architect for one of the five largest financial institutions in the U.S. for the past thirteen years. Frank holds a large number of professional certifications spanning a variety of technological areas and is a contributor to, founder and leader of a number of well-known open-source projects. Frank has authored a number of programming-related books and articles and regularly presents at various conferences and user group meetings. Frank founded a small software development company, Etherient, focused on mobile and cloud-based computing products for various platforms.